Modeling dynamic reconfigurations in Reo using high-level replacement systems

Krause, Christian; Maraikar, Ziyan; Lazovik, Alexander; Arbab, Farhad

Published in:
Science of computer programming

DOI:
10.1016/j.scico.2009.10.006

Science of Computer Programming 76 (2011) 23–36

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Modeling dynamic reconfigurations in Reo using high-level replacement systems

Christian Krause [a,*], Ziyan Maraikar [a], Alexander Lazovik [b], Farhad Arbab [a]

[a] Software Engineering Cluster, Centrum Wiskunde & Informatica (CWI), P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[b] Institute for Mathematics and Computing Science, University of Groningen, Nijenborg 9, 9747AG Groningen, The Netherlands

ARTICLE INFO

Article history:
Received 15 March 2008
Received in revised form 24 April 2009
Accepted 19 October 2009
Available online 25 October 2009

Keywords:
Dynamic reconfiguration
Graph transformation
Coordination

ABSTRACT

Reo is a channel-based coordination language, wherein circuit-like connectors model and implement interaction protocols in heterogeneous environments that coordinate components or services. Connectors are constructed from primitive channels and can be reconfigured dynamically. Reconfigurations can even execute within a pending I/O transaction. In this article, we formally model and analyze dynamic reconfigurations and show how running coordinators can be reconfigured without the cooperation of their engaged components.

We utilize the theory of high-level replacement systems to model rule-based reconfigurations of connectors. This allows us to perform a complex reconfiguration as an atomic step and analyze it using formal verification techniques. Specifically, we formalize the structure of connectors as typed hypergraphs and use critical pair and state space analyses for verification of dynamic reconfigurations. We provide a full implementation of our approach in a framework that includes tools for the definition, analysis, and execution of reconfigurations, and is integrated with two execution engines for Reo. Our framework, moreover, integrates with the graph transformation tools AGG and GROOVE for formal analysis, as well as the Eclipse platform and standard web service technologies.

1. Introduction

In component-based as well as service-oriented system models, the composition of the functional building blocks is a key task. The wide use of Internet standards, e.g. XML as a general-purpose, human readable data format and SOAP as a protocol for remote procedure calls and message passing has alleviated the challenges of distribution and heterogeneity. Apart from these rather technical issues, the main task, i.e., the correct and pragmatic coordination of the functional building blocks, remains unsolved. The problem with industry-proposed languages such as UML-2 or BPEL is their lack of formal semantics and compositionality. Moreover, these languages still often follow a low-level style of procedural programming languages. Coordination languages such as Linda [1] or Reo [2] are based on more abstract concepts, i.e., the notion of tuple space in the case of Linda and a channel-/connector-based coordination model in Reo, and they provide various means for verification and analysis. We follow this branch of research on coordination languages and study dynamic changes, i.e., reconfigurations of Reo connectors in this article. In a nutshell, we present a model for reconfigurations in Reo based on graph-rewriting techniques. We model Linda's tuple space notion as a dynamically reconfigurable Reo connector and analyze it using formal verification techniques.

* Corresponding author.
E-mail addresses: c.krause@cwi.nl (C. Krause), z.maraikar@cwi.nl (Z. Maraikar), a.lazovik@rug.nl (A. Lazovik), farhad@cwi.nl (F. Arbab).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.10.006

The need for dynamic reconfigurations does not exclusively arise in highly adaptive environments like peer-to-peer networks or multi-agent systems. Even a simple change in a coordination protocol, in a component's interface or its behavioral or QoS-properties can cause structural changes. From an engineering standpoint, of course only the predictable causes are interesting. However, even with a fixed set of possible event types, switching between a finite set of system configurations to accommodate the needs accordingly is not feasible. A rule-based approach for reconfigurations seems to be the only way to deal with the great complexity of dynamical structural changes in large systems. Moreover, atomicity of complex reconfigurations must be ensured, since they are usually performed as a series of low-level operations on primitives, e.g. a creation of a channel in Reo. Furthermore, when performed dynamically (at runtime) currently executing actions must be suspended or alternatively the reconfiguration has to be delayed until all pending actions are finished. An even bigger challenge in dynamic reconfigurations is to ensure a consistent system state after a reconfiguration, i.e., that no behavioral invariants get violated through the structural change. All these requirements are crucial in dynamically reconfigurable systems and there are no canonical solutions for them. For the case of the coordination language Reo, we propose a framework in this article that provides theoretical and practical means (formalisms and implementations, respectively) for the definition, analysis and execution of dynamic reconfigurations.

We propose an approach for defining reconfigurations for Reo using graph-rewriting techniques in this article. Specifically, we use the theory of high-level-replacement (HLR) systems, which can be seen as an extension of the algebraic graph transformation theory to other high-level structures. For an extensive introduction to algebraic graph transformation and HLR systems, we refer to [4]. The general idea is to define a set of rewrite rules, each consisting of a pattern, which must be matched, and an associated template, which describes the changes to be performed to the system. Additional application conditions (positive or negative) may restrict the transformation further. A major advantage of this rewriting approach is that it allows to specify abstractly in which situations and how a system should be changed, including possible dependencies that need to be updated. Further, these rules cannot only be applied locally, but also globally, i.e., wherever the patterns match. Another important aspect of this approach is the granularity of changes that are made. Instead of sequentially performing low-level modifications on primitives, complex structural reconfigurations can be achieved in an atomic step. In this article, we apply the ideas of HLR systems to the coordination language Reo and show how they can be used to model dynamic reconfigurations of Reo connectors. The dynamic aspect is in particular challenging because it involves both structure and semantics. This is also the reason why most existing work focuses either on providing

(1) models for high-level, rule-based reconfigurations (e.g. in [6]), or

(2) methods for reasoning about low-level reconfigurations (e.g. in [8]).

In the present article, we attempt to provide both high-level rule-based reconfiguration models as well as means for reasoning about them. We believe that this is crucial when studying dynamic reconfigurations.

Contributions. Concretely, we make the following contributions in this article. We provide a rule-based and high-level model for reconfigurations of Reo connectors. Specifically, we model connectors as typed hypergraphs and use algebraic graph-rewriting techniques. We use critical pair analysis for static verification of reconfiguration rules. As our running example, we model a coordination protocol that is parametrized on an unbounded number of components and resources. The reconfigurable connectors that we use for this example guarantees mutually exclusive access of n worker components to m resources. We then extend the example further and show how it naturally models a tuple space in Reo.

Further, we discuss how reconfigurations are performed on a deployed and running connector. In particular, we discuss how reconfigurations can be executed without the cooperation of the engaged components, and even in the middle of an I/O transaction. Moreover, we address the problem of ensuring structural and behavioral invariants after performing dynamic reconfigurations by a careful design of the reconfiguration rules, which in particular includes state information of the engaged channels and components.

We provide a full implementation of our reconfiguration approach for Reo, including tools for defining, verifying and executing dynamic reconfigurations. For the definition, we provide graphical editors. For verification, we have implemented conversion tools that produce output for the Attributed Graph Grammar (AGG) system [4,11] and the GROOVE tool [12]. Using these tools, we can perform confluence and termination checks for reconfiguration rules as well as state space exploration and model checking of dynamic reconfigurations. By integrating our reconfiguration tools with two execution engines for Reo and standard web service technologies, we provide a proof of concept implementation of a dynamically reconfigurable coordination web service. The whole framework is built on top of the Eclipse development platform.[1]

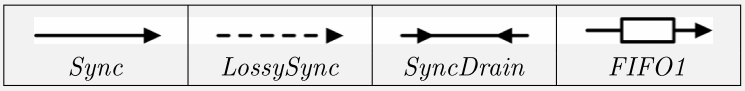
Tools overview. Tool support will play a crucial role throughout this article, which is why we have decided not to have a single tool section, but to discuss the different parts of our framework in each section separately after having introduced the necessary theory. The list of tools used in this article is given in Table 1. On one hand, ECT as well as ReoLive are implementations of Reo. On the other hand, AGG and GROOVE are third-party graph transformation tools that we use for analysis.

[1] Eclipse homepage: http://www.eclipse.org.

Table 1
List of used tools.

Tool name	Provider
Eclipse Coordination Tools (ECT) [14]	CWI
ReoLive coordination web service [15]	CWI
The Attributed Graph Grammar system (AGG) [11]	TU-Berlin
GRaphs for Object-Oriented VErification (GROOVE) [12]	University of Twente

Table 2
Four basic channel types.



Organization. The rest of the article is organized as follows. In Section 2, we describe the language concepts of Reo, introduces a small set of basic channel types and our running example, and gives a brief overview of the Eclipse-based tools for Reo. In Section 3 then, we introduce the basic concepts of HLR systems and show how they can be applied to Reo for modeling reconfigurations. Furthermore, we describe the implementation of a reconfiguration engine and a conversion tool for verification of reconfiguration rules. In Section 4, we discuss the dynamic aspects of reconfigurations including a formal analysis of the dependencies of reconfigurations and executions, and a proof of concept implementation of a dynamically reconfigurable coordination web service. Finally, Section 5 contains conclusions and in Section 6 we give an overview of related work.

2. Coordination with Reo

In the first part of this section, we introduce the notion of connectors and recall the basic language features of Reo. We introduce a small set of channels, which we will use later in our running example. In the second part, we give an overview of our tool support for Reo. We discuss briefly two formal semantics of connectors, which form the basis for our formal analysis tools as well as for execution platforms for Reo.

2.1. Language overview

Reo [2] is a channel-based coordination language. Channels are primitive entities with well-defined behavior, supplied by users. They can be arranged into circuit-like connectors, which implement certain protocols and thereby coordinate active entities, such as components from outside (hence, referred to as exogenous coordination). Channels impose constraints on the dataflow at their ends. In particular, they can synchronize or mutually exclude dataflow. They can further contain buffers or have context-dependent behavior [9,10]. In Reo, all these properties are compositional, i.e., they carry over to the level of connectors.

For examples in this section, we use the set of channels depicted in Table 2. The *Sync* channel consumes data items at its source end and dispenses them at its target end. The I/O operations are performed synchronously and without any buffering. Consequently, the channel blocks if the party at the target end is not ready to receive any data. The *LossySync* channel behaves in the same way, except that it does not block the party at its source end. Instead, the data item is consumed and destroyed by the channel. The *SyncDrain* channel is also a synchronous channel, but it differs in the fact that it has two source ends through which it consumes and destroys data items synchronously. The last channel that we consider here is the *FIFO1*. It is an asynchronous, directed channel with a buffer of size one.

To compose channels, Reo provides a pre-defined and fixed notion of nodes. Nodes merge incoming data items and replicate them to all outgoing channel ends. This can be seen as a 1:*n* synchronization, as opposed to 1:1 synchronizations (Milner style), or synchronizations of all coinciding channel ends (Hoare style). Note also that nodes do not buffer data items. Reo further distinguishes between public and internal nodes, also referred to as boundary and mixed nodes, respectively. In our examples, we will depict the former as empty and the latter as filled circles.

Example 1. Fig. 1 depicts a Reo connector that coordinates three components. In this case, we consider simple reader and writer components which produce and consume data through a port. In the literature, this connector is usually referred to as exclusive router (cf. [3]). It routes data synchronously from the writer at node *A* to only one of the readers at nodes *C* and *D*. If both readers are willing to accept data, the choice of where the data item goes is made non-deterministically. This is due to the fact that node *B* merges its inputs without priority, i.e., exactly one of the *Sync* channels is activated, the data item on the active side is replicated to the corresponding reader and the data item on the other side is destroyed by its corresponding *LossySync*.

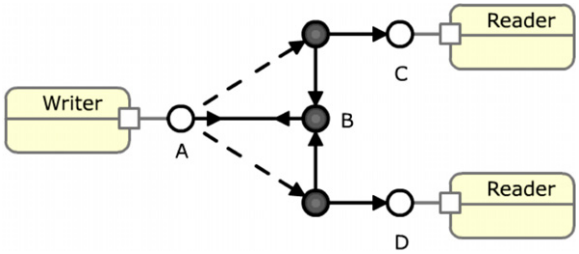


Fig. 1. Exclusive router.

Example 2. Fig. 2 shows another well-known connector: the ordering or alternator (cf. [2]). Again, we use simple reader and writer components for illustration. The protocol imposed by this connector is an ordered output of the data items provided by the two writers at nodes *A* and *B*. The *SyncDrain* is used to synchronize the inputs. The *FIFO1* stores the data item from *B* and makes it available in the next step. Since the *FIFO1* cannot store more than one data item, it has to be released first before new data items can be read.

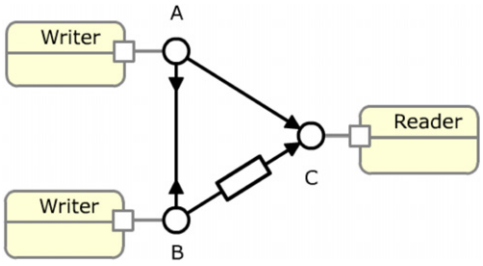


Fig. 2. Ordering.

Example 3. Finally, the connector in Fig. 3 constitutes our running example. In this connector, the components are no longer reader and writers. Instead they model abstract worker units that require mutually exclusive access to a resource. The token in the *FIFO1* channel models this resource. Moreover, we use the symbol ⊗ to denote the exclusive router from our first example. Note that the worker components now have two ports for communication: one for reading the resource and start processing, and one for releasing the resource again after the processing is finished. Later in this article, we extend this example further and define reconfiguration rules for it.

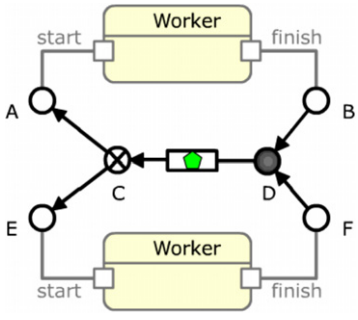


Fig. 3. Mutually exclusive access to a resource.

In the language reference for Reo [2], channels are described as mobile entities. Their ends (and thereby the nodes they connect) can transparently be moved to other physical locations. More relevant to the context of this article is the fact that channels can be instantiated and destroyed, and nodes can be joined or split at runtime. In essence, Reo provides low-level operations for dynamically reconfiguring connectors. It is the aim of this article to provide a framework for the definition and dynamic execution of complex reconfigurations, based on graph-rewriting techniques.

In the following, we briefly introduce two semantical models for Reo, which are important for verification and execution of connectors.

2.2. Semantical models

For the scope of this article, two semantical models for Reo are interesting: constraint automata [3] and the coloring semantics [9]. Both models are compositional in the sense that the semantics of a connector is computed from the semantics of its constituent primitives. Furthermore, both models equivalently represent synchronization and mutual exclusion constraints imposed by channels. While in constraint automata the focus is on state and dataflow, the coloring semantics focuses on the context-dependent behavior of individual primitives, such as the *LossySync* channel or priority mergers. Note also that in a recent work [10], some effort has been made to extend the constraint automata model with context-dependency and thereby integrating it with the coloring semantics.

The constraint automaton for our running example is shown in Fig. 4. Without going into the details, it is worth mentioning that in this example we have hidden the internal nodes C, D and that the unary buffer of the *FIFO1* is modeled using a state memory cell x. The automaton has three states, respectively, representing the resource being in the *FIFO1*, or allocated to one of the two workers.

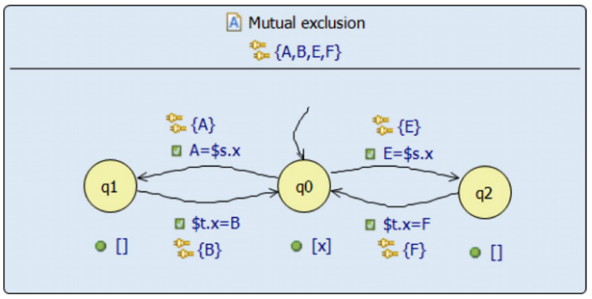


Fig. 4. Constraint automaton for the running example.

The basic idea of the coloring semantics is to associate dataflow colors to channel ends. A coloring of the running examples is shown in Fig. 5. Thick, blue lines indicate synchronous dataflow. The red arrows in the no-flow regions can be interpreted as exclusion constraints, which are used to rule out illegal behaviors of context-dependent primitives. An extension of the coloring model also incorporates dataflow actions, such as creating, moving or destroying data items. Using this extension, we can compositionally compute animations of connectors. In fact, Fig. 5 is also a snapshot of an animation where the resource is being allocated to the bottom worker.

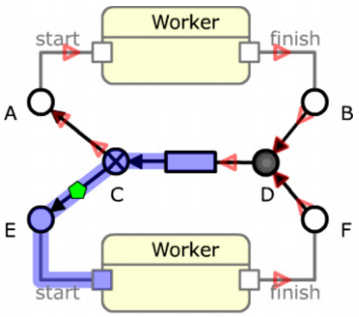


Fig. 5. Coloring/animation of the running example.

Based on these two semantics of connectors, we have implemented a number of tools. In particular, the constraint automata model is used in a model checker and a centralized execution engine, and the coloring semantics is the theoretical underpinning of an animation tool and a distributed execution engine for Reo, which we discuss in some more detail in Section 2.3.

2.3. Tool support: Eclipse coordination tools

We provide strong tool support for Reo within the Eclipse Coordination Tools (ECT) project [14]. This framework consists of a set of integrated plug-ins for the Eclipse platform. It is mainly developed by the SEN-3 group at CWI and includes among others the following tools:

- a structural model and a graphical editor for Reo connectors,
- an animation tool and a model checker for verification, and
- a centralized and a distributed execution engine for connectors.

Note that all the examples in this article were modeled using the graphical Reo editor. Moreover, we have a full implementation of the constraint automata and coloring semantics. In the following, we briefly discuss the verification tools and the execution engines. The latter are of particular relevance for us, since the reconfiguration to be defined later will be dynamically executed in these engines.

Verification. The two most important verification tools are an animation plug-in and a symbolic model checker [16], which is developed by the group of Christel Baier at the Technical University of Dresden. The animation tool provides a fully automated generation of connector animations, based on the coloring semantics. The model checker uses a branching time logic for specifications and is available as a standalone tool and integrated into ECT.

Execution. There are two execution engines available for Reo: a centralized and a distributed one. In the centralized version, connectors are statically compiled into a constraint automaton which is directly executed, either by an interpreter or by generated code. The automaton specifies the set of ports that must be active to trigger a transition from a state. Once a transition is triggered, data are transferred between ports atomically, as specified by the transition constraint. On the other hand, the distributed approach allows primitives belonging to a connector to be deployed across multiple engines. This is achieved by *splitting* nodes across engines. A distributed protocol guarantees that all parts of the connector agree on its next behavior, ensuring the synchrony that is specified by the connector. This provides a scalable implementation with better support for reconfiguration by decoupling the execution of independent parts of the connector. Both engines support external components that communicate with the engine through synchronous `read` and `write` operations on *ports* of the executing connector.

ReoLive [15] is a wrapper around Reo execution engines, which exposes their capabilities as web services. At present, ReoLive utilizes the centralized engine, but we plan to adapt it to the distributed engine as well. The ReoLive service provides a web-based user interface that lets users upload connectors created using the graphical Reo editor. Users may start and stop connector instances, and perform reconfiguration operations. ReoLive's programmatic interface exposes the ports of the Reo connector being executed by an engine as web service end-points. This allows the coordination of remote components that communicate through simple `read` and `write` operations on these web service end-points. Moreover, ReoLive supports transparent reconfigurations. Before explaining the details of this approach, we introduce first the necessary theory for graph-based reconfigurations in the next section.

3. Connector rewriting

In this section, we formalize the structure of connectors using typed hypergraphs, a high-level structure that extends the notion of simple graphs. We show further how reconfiguration rules are formally defined and applied using HLR systems. We apply the technique of critical pair analysis to verify local confluence of reconfiguration rules. On the implementation side, we present our tool support for reconfigurations, which follows exactly the used formalisms. An integration with the graph transformation tool AGG [4,11] enables us to perform critical pair analysis.

3.1. High-level replacement systems

The theory of graph transformation goes back to 1970s and has its roots in Chomsky grammars on strings, and in term—and effectively tree—rewriting. HLR systems can be seen as an adaptation of the algebraic graph-rewriting techniques to high-level structures, such as labeled graphs, typed graphs, hypergraphs, attributed graphs, Petri nets and algebraic specifications.

The core concept in HLR systems is rule-based rewriting of structures, as depicted in Fig. 6. Rewrite rules are defined in terms of a left-hand side (LHS), a right-hand side (RHS) and a mapping between these two. An application of a rule to a source structure results in a modified target structure. Note that this modification can be performed *in-place*, i.e., the source structure is modified directly. This is an important criterion for applicability in dynamic reconfiguration scenarios, since at runtime it is in most cases not desirable to redeploy the complete system. Note also that a single rule can describe complex reconfigurations, which are performed in an atomic step. This is a major advantage of graph rewriting over conventional approaches for reconfiguration, where a series of low-level operations must be performed and the consistency of the result has to be ensured through additional mechanisms. Moreover, rules may be extended with (negative) application conditions, for instance to make reconfiguration state- or context-dependent. If a single rule is not sufficient, a grammar consisting of multiple rules, which can be applied using additional control-flow mechanisms can be used. The expressive power of HLR systems is more than suitable to describe reconfigurations.

Regarding formal analysis, the most important technique for HLR systems is the so-called critical pair analysis, which is used to check local confluence. The technique originates again in term rewriting. A generalization to hypergraphs was first considered in [17] and to typed node attributed graphs in [18]. A critical pair is a pair of applications of two rules to the same structure that are in conflict. The most common reason for a conflict is that one rule deletes an object which is matched by the other rule (*delete–use* conflict). Essentially, in a critical pair, one rule application disables the other rule application. In the reconfiguration domain, critical pair analysis is a useful tool for reasoning about sets of reconfiguration rules and it can help to eliminate design failures. We also note that the HLR theory provides methods for verifying termination as well.

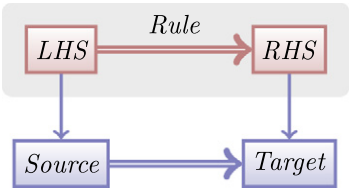


Fig. 6. Rule-based rewriting.

However, in our applications this plays a less important role. Yet another way of verifying properties of reconfigurations is the possibility of simulating rule applications. We will use this technique (state space exploration and bounded model checking) later when we discuss dynamic reconfigurations.

3.2. Connectors as typed hypergraphs

An appropriate structural model for Reo connectors are typed hypergraphs. Nodes are represented by vertices and channels by (hyper-) edges in this model. Note that a hypergraph model can capture the notion of undirected, i.e., drain and spout, channels directly. An encoding into ordinary graphs with directed edges is in principle possible, but it is not a natural representation of this concept. Moreover, hyperedges are an appropriate model for components in our framework. Components have a fixed interface consisting of a number of source and target ends, and they are treated exactly like channels. The following definition formalizes the notion of hypergraphs.

Definition 1 (*Hypergraph*)**.** A hypergraph $G = (V, E, s, t)$ consists of a set of vertices V, a set of edges E and source and target functions $s, t{:}E \to V^*$.

Definition 2 (*Hypergraph Morphism*)**.** A hypergraph morphism $f{:}G_1 \to G_2$ is a pair of functions $f = (f_V, f_E)$ with $f_V{:}V_1 \to V_2$ and $f_E{:}E_1 \to E_2$, such that $f_V^* \circ s_1 = s_2 \circ f_E$ and $f_V^* \circ t_1 = t_2 \circ f_E$.

Remark 1. With $(-)^*$ we denote the free monoid functor, which takes a set A to the set A^* of finite words over A.

Hypergraphs and their morphisms form a category, denoted by **HGraph** in the following. As pointed out above, hypergraphs naturally model the structure of connectors. However, for a complete representation, a typing mechanism must be in place to differentiate between multiple channel and component types. For this purpose, we consider hypergraphs together with morphisms into a fixed typegraph, denoted by **Reo**. This typegraph defines all allowed types of nodes, channels and components. Let $(G_i, type_i{:}G_i \to \mathbf{Reo})$ for $i \in \{1, 2\}$ be two Reo graphs. Then, a morphism $f{:}G_1 \to G_2$ is an ordinary hypergraph morphism that additionally satisfies the equation $type_2 = f \circ type_1$. We denote the category of Reo graphs by $\mathbf{HGraph_{Reo}}$. Hence, our final structural model for Reo connectors are typed hypergraphs.

Remark 2. In all our examples, we use the symbols introduced in Section 2 to denote the type of the channels. The type of components is indicated by their names.

Example 4. Recalling the connector of our running example in Fig. 3, its underlying hypergraph consists of six nodes (A–F) and in total seven hyperedges, modeling five channels and two components of type *Worker*. The exclusive router labeled with C is modeled using a second node type. However, we can also use the symbol $\otimes$ as an abbreviation for the hypergraph of the actual exclusive router in Fig. 1. Both encodings are valid models in our context.

3.3. Double pushout rewriting

The most well-studied way of defining and applying transformations for high-level structures like our typed hypergraphs is the double pushout (DPO) approach. Although the DPO approach is generic and can be applied to many types of high-level structures, in the sequel we consider its application to connectors only.

In the DPO approach, rewrite rules are formally defined as spans of injective morphisms $p = L \xleftarrow{\ell} K \xrightarrow{r} R$ in the category $\mathbf{HGraph_{Reo}}$. The LHS L defines the structural pattern that must be matched to apply the rule. The so-called *gluing* connector K contains all elements that are not removed by the rule and R additionally has those elements of the connector that are created by the rule.

To reconfigure a connector M using a rule p, we need to define a morphism $m{:}L \to M$, called *match*. The reconfiguration of M using the rule p with the match m is formally defined as the following diagram where (1) and (2) are pushouts.

$$\begin{array}{ccccc} L & \xleftarrow{\ell} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ M & \xleftarrow{} & C & \xrightarrow{} & N \end{array}$$

Operationally, the connector M is transformed to the connector N by (i) removing the occurrence of $L\backslash\ell(K)$ in M, yielding the intermediate connector C, and (ii) adding a copy of $R\backslash r(K)$ to C. This is the core of the DPO approach and it works for a number of high-level structures.

In the rest of the article, we depict reconfiguration rules just by their LHS and RHS. The gluing connector K is defined as the intersection of the two connectors and the mappings ℓ and r are implicitly given by the labels and the relative positions of the nodes and edges.

Example 5. Fig. 7 depicts the reconfiguration rule *AddWorker* for the mutual exclusion connector. The rule creates a new worker component and wires it to the connector. Note that it is still valid to consider the node C not as a vertex in the graph but just as an abbreviation for the exclusive router. This is possible because we can define, without much effort, a reconfiguration rule that adds another outgoing end to an exclusive router. Note that we can also specify a rule that takes an existing worker and just wires it to the connector.

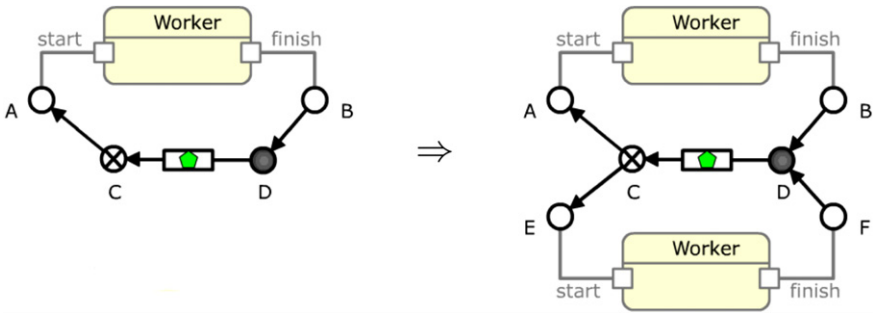


Fig. 7. Reconfiguration rule: *AddWorker*.

Example 6. So far, only a single resources was managed by the connector. To allow more than one resource, we define the second reconfiguration: *AddResource*, depicted in Fig. 8. It basically adds another *FIFO1* to the connector, which carries the new resource. However, we also have to change the original connector, turning node D into an exclusive router. This is required because otherwise the resource would be replicated by D and send to all *FIFO1* buffers. Note also that we abstract from the actual content of the resource. We assume that when this rule is applied to a real Reo network, the resource is given as a parameter to the rule, e.g. using a URI.

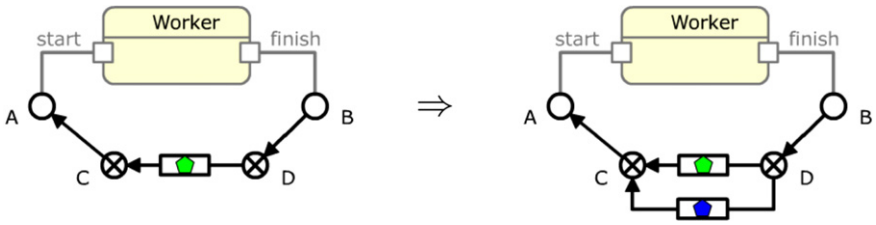


Fig. 8. Reconfiguration rule: *AddResource*.

The presented reconfiguration rules enable us to dynamically add workers and resources to the basic connector presented in Section 2. The reconfigurable connector now effectively ensures the mutually exclusive access of n workers to m resources, a non-trivial coordination pattern.

We can extend the example further by assigning data constraints to the outgoing channels of C (recall that channels are user-defined entities in Reo). Such a data constraint can be used to define a pattern that the resource must match in order to be suitable for processing by a specific worker. For instance, the resource can be given as URI. If the workers can understand only certain protocols, such as *http*, *ftp*, etc., the data constraint can ensure that the scheme of the URI matches that protocol. Moreover, we can also consider a scenario where the workers are allowed to modify the resource (either directly its data or the pointer that references the actual data). To that end, the system that we have modeled effectively describes a tuple space, which is a type of distributed shared memory and the central concept in the coordination language Linda [1].

Having introduced the necessary theory for modeling and applying reconfiguration rules, we now discuss our tool support.

3.4. Tool support: Reconfigurations in ECT and AGG

We have implemented an adaptation of the DPO approach for Reo connectors in ECT, consisting of a reconfiguration model, a reconfiguration engine for finding matches and applying rules, and tree-based editor for the defining reconfiguration rules. Moreover, the tools allow to automatically derive rules from LHS–RHS connector pairs and to test them in the graphical editor.

We decided to implement the reconfiguration framework from scratch, instead of using third-party tools for graph transformation, such as AGG. Our hypergraph model is not supported by most existing graph transformation tools. Although an encoding into ordinary graphs is possible, it yields a too low-level representation that does not reflect our model naturally. In our implementation, reconfiguration rules are natively defined over the existing Reo model.

For analysis, we have implemented a conversion tool to export reconfiguration rules into the AGG format for graph grammars. In particular, we can use AGG's built-in critical pair analysis tool. Fig. 9 depicts the exported version of the rule *AddWorker* in AGG. Note that channels and components are no longer represented by hyperedges, but by nodes. To test the critical pair analysis, we have included a third rule to our example: *DelResource*, which is just the inverse of *AddResource*. The output of the critical pair analysis in AGG is shown in Fig. 10, depicting that we have verified that *DelResource* disables all other rules and itself. The reason for this is that the deleted resource could have been matched by the other rules. The rules *AddWorker* and *AddResource*, on the other hand, do not conflict with each other, i.e., they are *parallel independent* (cf. [4]) and can thereby be applied in any order with the same result.

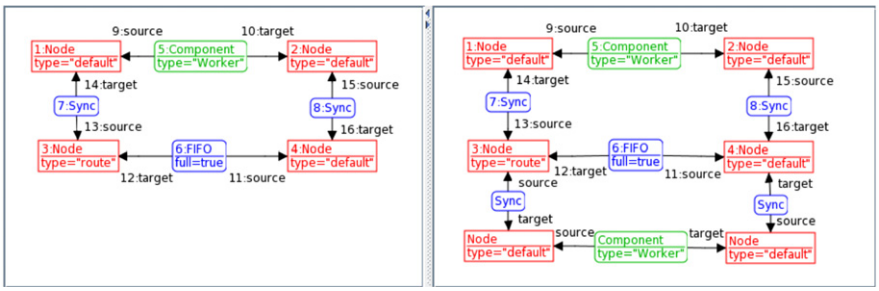


Fig. 9. Reconfiguration rule *AddWorker* in AGG.

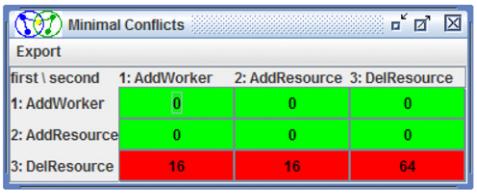


Fig. 10. Results of critical pair analysis in AGG.

4. Dynamic reconfigurations

Dynamic reconfigurations are reconfigurations of deployed and running connectors, which we often refer to as *coordinators*. In this section, we discuss two topics: one regarding formal analysis and one regarding the execution of dynamic reconfigurations. In the first part, we illustrate how structural and behavioral invariants can be used to ensure consistency of dynamically reconfigurable systems. In the second part, we discuss how a coordinator actually can be reconfigured during an execution and even within a pending I/O transaction. We conclude this section with a brief description of a proof of concept implementation of a dynamically reconfigurable coordination web service.

4.1. Ensuring structural and behavioral invariants

Structural and behavioral invariants are properties of dynamic systems that can be used to ensure their consistency constraints, given e.g., as part of the requirements or the specification of the system. Since they modify the structure of a system, reconfigurations must be designed such that structural invariants are not violated. Moreover, when applied at runtime, behavioral invariants can also be affected. In the following, we illustrate the use of invariants on our running example. We consider the following two invariants for the tuple space connector:

(1) There is at least one worker and one *FIFO1* buffer for a resource.

(2) The number of empty *FIFO1*s equals to the number of active workers.

The first invariant is purely structural. The second one makes an assertion about the states of the *FIFO1*s and the worker components and is thereby of structural and behavioral nature. It ensures that there are not more resources being used than can be stored in the *FIFO1*s. We assume that initially there is one idle worker and one *FIFO1* in the system. Both invariants hold at this stage.

Invariant (1) can get violated when resources or workers are deleted. So far, we have only considered the first case using the rule *DelResource*, which was defined as the inverse of *AddResource* in Fig. 8. The LHS of *DelResource*, in fact matches two *FIFO1* buffers and deletes only one of them. If there is only one *FIFO1*, the rule cannot be applied (we assume injective matches). Accordingly, a rule *DelWorker* can be defined that does not violate the first invariant.

Invariant (2) is more complex, since it involves also the states of the *FIFO1*s and the worker components. However, if we look at the rule *AddResource* it is evident that it does not violate the constraint. This is due to the fact that it creates a new FIFO1 buffer together with a new resource, i.e., the *FIFO1* is initially full. Moreover, *DelResource* also respect the invariant. However, we cannot verify if adding and removing workers violates the invariant, because we cannot make any assumption about the states of the workers. Our solution for this problem is to explicitly model the state of a worker as a boolean attribute, called *active*. Attribution is a commonly used and well-studied concept for extending graph-rewrite rules with algebraic structure [5,4]. Fig. 11 depicts the refined rule *AddWorker* that explicitly sets the *active*-flag of the newly created worker to `false`. Now, we can actually verify that the rule does not violate the second invariant (the state of the upper worker is preserved by the variable *x*). Accordingly, we can define the rule *DelWorker* as the inverse rule and thereby guarantee that only inactive workers are deleted.

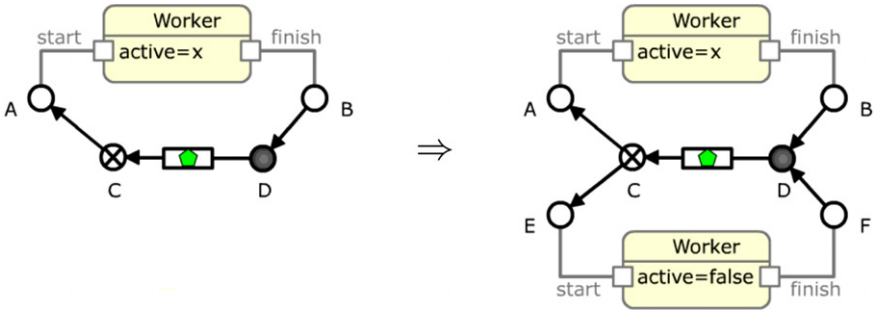


Fig. 11. Refined reconfiguration rule: *AddWorker*.

We believe that including state information in the form of attributes in reconfiguration rules is a proper way to ensure both structural as well as behavioral invariants. A careful design is in any case crucial for dynamic reconfigurations. However, what is left is a formal verification of invariants. A possible way to do this is to formalize the execution of a connector also as rewrite rules and to analyze them together with the reconfiguration rules in a model checker for graph grammars. In general, this is a non-trivial task. In particular, compositionality of such a rewrite rule semantics for connectors is very hard to achieve. For our example, however, we can define such rules directly. Fig. 12 depicts the rule *StartWorker*, which models the action of assigning a resource to a worker and starting it. The rule *StopWorker* can be defined as its inverse. Now, we have defined the complete dynamic behavior (execution and reconfiguration) of the system using graph-rewrite rules. We can thereby also analyze the system formally using graph transformation tools, which we will discuss in the next section.

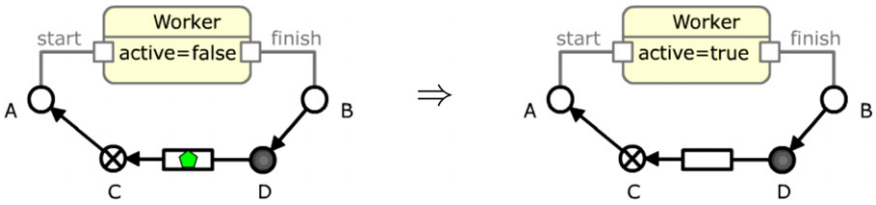


Fig. 12. Execution rule: *StartWorker*.

4.2. Tool support: State space analysis in GROOVE

GROOVE [12] is a tool for state space analysis and model checking of graph grammars. Its main application areas are model transformations and operational semantics. Its support for state space exploration and model checking makes it very suitable for analyzing dynamic reconfigurations.

We have implemented a conversion tool that generates graph grammars for GROOVE from Reo reconfiguration rules. We have tested the tool using the reconfiguration rules for adding and deleting workers and resources, and moreover for starting and stopping workers, as presented in the previous section.

Fig. 13 depicts a manual state space exploration in GROOVE. The initial state $s1$ is the system with one inactive worker and one resource. In Section 3.4, we have verified already that the rules *AddWorker* and *AddResource* are parallel independent. This is also true for the modified version of *AddWorker*. In the state space exploration, the parallel independence causes the typical diamond shape comprised of the states $s1$–$s4$. An interesting path is $s2 \rightarrow s5 \rightarrow s6 \rightarrow s3$, where a worker is started, another worker is added and the first worker is stopped again. Since *StopWorker* is the inverse of *StartWorker*, the resulting state is the same as that one would get if only *AddWorker* were applied. Another interesting fact is that the labeled transition system is asymmetric at this point. One might expect that there exists a path between $s7$ and $s3$ through *AddResource* and *StopWorker*. At second glance, it becomes evident that *AddResource* is not enabled in $s7$, because it requires an existing resource to be applicable. While in $s2$ a second resource that can be used has been created in the previous step, in $s7$ the only existing resource is already allocated to a worker.

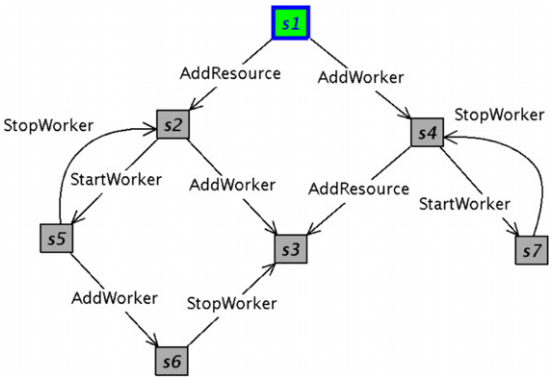


Fig. 13. Manual state space exploration in GROOVE.

The GROOVE tool set also includes support for model checking [13]. In the tool, properties are specified using CTL [19] formulas. Since the labeled transition generated by our rewrite rules is infinite, we have to set an upper bound for the model checking. We chose a limit of 10 workers and 10 *FIFO1*s. The resulting transition system has approximately 470 states and 12 000 transitions. We have successfully verified invariant (1) and (2) in this setting. The second invariant required essentially a counting of full *FIFO1*s and active workers, which can be encoded using additional rules and the notion of priority in GROOVE. Further, we have tested the following properties:

(1) `AG(StartWorker → EX(StopWorker))`: After starting a worker it can always be stopped again in the next step. The property is satisfied.
(2) `AG(AddResource)`: It is always possible to add a resource. This property is not satisfied, as mentioned above already. The same holds for *AddWorker*.
(3) `AG(StopWorker → EX(AddResource))`: After stopping a worker it is always possible to add a new resource in the next step. The property is satisfied.

Particularly, the last property combines both dynamic structural and behavioral system properties. As a consequence of the state space analysis, we are now able to fine-tune our reconfiguration rules by defining the rules for adding workers and resources irrespectively of the state of the existing *FIFO1*, thereby ensuring that these dynamic reconfigurations can always be applied.

This ends our discussion on the analysis of dynamic reconfigurations in Reo. We have used three different techniques to analyze properties. We have checked parallel independence of rules using the critical pair analysis in AGG. Then, we have manually simulated reconfigurations together with executions in GROOVE and verified the simulation results using a model checker. These techniques on hand, we can now discuss further how an actual implementation of a Reo connector, i.e., a deployed and running coordinator can be adapted to support dynamic reconfigurations.

4.3. Reconfigurations within transactions

As pointed out in Section 2, synchronizations across a connector are an integral part of the Reo semantics. If this is seen in the context of executions of connectors, it corresponds to performing I/O operations in transactions, which is part of both, the centralized and the distributed implementation (cf. Section 2.3). Moreover, both execution engines provide a common port abstraction for component–coordinator communication. Therefore, we use an abstract notion of coordinator to refer to one of the two implementations. We further assume that such a coordinator can be reconfigured, without actually fixing

how the reconfiguration is performed. We will later discuss how this is actually done in the two existing implementations of Reo runtime systems.

In the process of a dynamic reconfiguration in Reo, the following parties are involved: a coordinator, a number of components that communicate with the coordinator through ports, a reconfiguration engine that performs DPO rewriting on the coordinator, and an administrator that initiates the reconfiguration. Fig. 14 contains a message sequence chart that shows how a reconfiguration is performed within a pending transaction. Note that the component is oblivious to the reconfiguration and that the reconfiguration engine does not need to wait until the current transaction is finished. The sequence of actions performed is as follows:

(1) An administrator initiates the reconfiguration using a given rule and a corresponding match into the deployed and running connector.
(2) The reconfiguration engine interrupts all pending I/O operations on the coordinator. All ports are suspended.
(3) The reconfiguration engine reconfigures the connector using the given rewrite rules and resumes the coordinator when finished.
(4) All ports are resumed and pending I/O operations performed on the reconfigured coordinator.

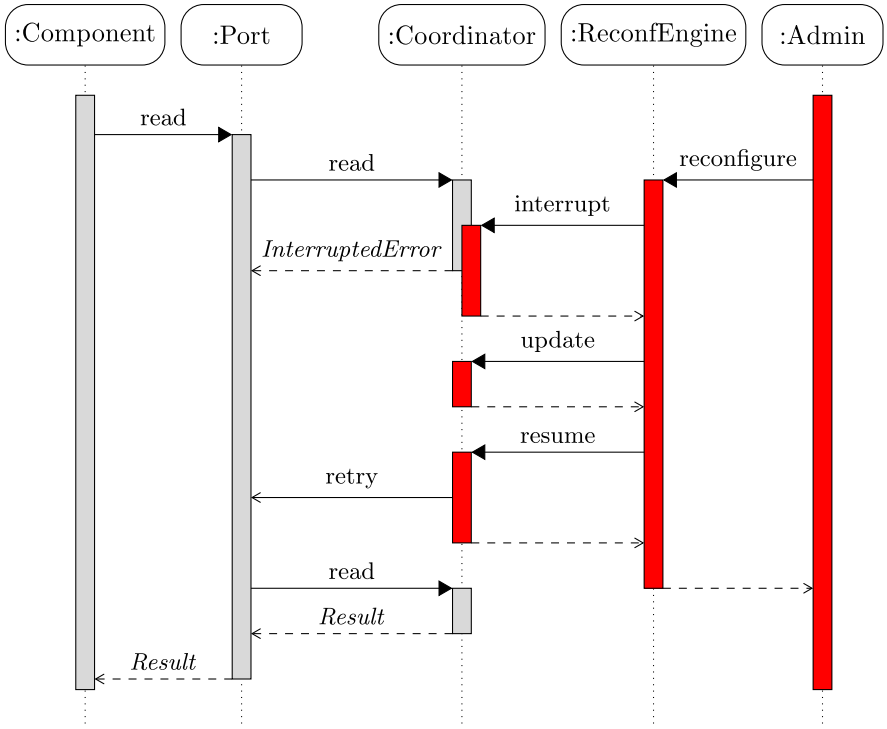


Fig. 14. A reconfiguration within a pending transaction.

In the sequence chart, the actual reconfiguration action is abstracted in the `update` operation that the reconfiguration engine performs on the coordinator. Recall that in the centralized engine, the whole connector is compiled into a constraint automaton first, which is then directly executed as a state-machine. A reconfiguration in this case is performed only on the connector model, which is afterwards recompiled into a constraint automaton and restarted. However, the state of the coordinator must be restored in this case. To that end, the contents of memory cells (*FIFO1* buffers) as well as the actual current state in the automaton must be restored. The latter is non-trivial, because it cannot be expected that there is any correlation between the constraint automata before and after a reconfiguration. However, the best approximation is to restore the states of the primitives that are not changed by the reconfiguration and to use the default initial states of the newly created primitives. To implement this, one has to basically compute the projections from the automaton for the whole connector into the automata of the primitives. This ensures that the correct system state can be restored. Reconfigurations in the distributed execution engine are much easier to handle than in the centralized, constraint automata approach. In the distributed case, reconfigurations can be performed directly on a deployed connector. The problem of restoring the correct system state after the reconfiguration does not occur, since in fact the system state is distributed in this case.

Tool support. Transparent dynamic reconfiguration support has been integrated into the ReoLive service described in Section 2.3. The handling of suspending and resuming of transactions is implemented by ReoLive's web service port wrappers.

5. Conclusions

In this article, we have described an approach for dynamic reconfigurations of systems that are deployed in continuously changing environments, based on the concepts of the coordination language Reo. Connectors are represented as typed hypergraphs and reconfigurations are modeled as graph-rewrite rules in our approach. We applied a critical pair and state space analysis as well as model checking for verifying properties of dynamic reconfigurations. We have demonstrated the developed reconfiguration techniques on a running example dealing with resource mediation, which we later extended to a tuple space model in Reo.

One of the important issues with dynamic reconfigurations is preserving the state consistency of components and of the overall execution. We have shown that within our approach, reconfigurations may be safely performed on a deployed and running connector without the cooperation of the engaged components. Safety is achieved by using formal verification and model checking techniques, thus ensuring structural and behavioral consistency after performing reconfiguration rules. For verification, we export our connectors and rules to graph transformation tools AGG and GROOVE.

We have provided a reference implementation for the proposed dynamic reconfiguration approach, including tools for defining, verifying and executing reconfigurations. Our reconfiguration framework is integrated with existing execution engines for Reo, as well as with a web service protocol stack, thus providing an easy way to integrate the approach with existing systems working on top of Reo.

In this work, we have shown a high-level approach for modeling and verifying dynamic reconfigurations of Reo-coordinated systems. This brought up a number of new research directions that are left out of the scope of this article. For example, when reconfiguration rules are used to achieve some objectives, e.g. to satisfy a particular requirement by a reconfiguration, some rules cannot be applied together, since one rule may potentially neglect the effects of previously applied reconfigurations. This extends the notion of conflict used in the critical pair analysis. A possible way of dealing with such properties is to use AI techniques such as planning to find a sequence of reconfiguration rules that satisfies the provided objectives.

6. Related work

Graph-based models for Reo connectors and their reconfigurations have been considered in [6,7]. The first of the two articles deals with the models for distributed connectors and provides a mechanism to synchronize distributed reconfigurations. The second article discusses an approach where reconfigurations are triggered by dataflow events. The present article extends both works by a formal analysis and a full implementation of dynamic reconfigurations for Reo.

A basic logic for reasoning about connector reconfigurations, including a model checking algorithm is the topic of [8]. Different than the work in this article, the author uses a formalization of connectors, which is particularly not a graph model. Moreover, the reconfiguration operations are rather low-level and provide no means for a rule-based definition of reconfigurations.

In a more general setting, dynamic reconfigurations have been studied for a number of Petri net variants. For instance, open Petri nets [20] extend the original model by a notion of open places, representing an interface to their environment. As in our approach, reconfigurations of open Petri nets are modeled using DPO rewriting. Behavior preserving reconfigurations of open Petri nets are studied in [21].

A typical application of open Petri nets are workflow nets [22]. Dynamic changes in workflow nets are discussed in [23]. The so-called *dynamic change bug* refers to a problem of ensuring a consistent system state after a reconfiguration. This has some similarities to the notion of structural and behavioral invariants in this article. While in our approach invariants are application specific and basically user-defined, the consistency for workflow nets ensures only a basic notion of correctness, i.e., a reconfiguration is valid if the state after the reconfiguration could have been reached from the initial state. Their proposed solution is to calculate a safe change region and to allow reconfigurations only in these states.

Architectural Design Rewriting (ADR) [24] is a framework for modeling reconfigurable software architectures. This approach uses hyperedge replacement for defining hierarchical structures. Contrary to our approach, term rewriting is used for reconfigurations in ADR. Generally, the focus in ADR is on style-preservation, i.e., structural (and not behavioral) properties of reconfigurations.

Acknowledgement

First author's work was supported by NWO GLANCE funding programs *WoMaLaPaDiA* and *SYANCO*.

References

[1] D. Gelernter, N. Carriero, S. Chandran, S. Chang, Parallel programming in linda, in: International Conference on Parallel Processing, ICPP, 1985, pp. 255–263.
[2] F. Arbab, Reo: A channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (2004) 329–366. doi:10.1017/S0960129504004153.
[3] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in Reo by constraint automata, Science of Computer Programming 61 (2) (2006) 75–113. doi:10.1016/j.scico.2005.10.008.

[4] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, EATCS Monographs in Theoretical Computer Science, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[5] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories, Fundamenta Informaticae 74 (1) (2006) 31–61.
[6] C. Koehler, F. Arbab, E. de Vink, Reconfiguring distributed Reo connectors, in: Recent Trends in Algebraic Development Techniques, WADT'09, in: Lecture Notes in Computer Science, vol. 5486, Springer, 2009, pp. 221–235. doi:10.1007/978-3-642-03429-9.
[7] C. Koehler, D. Costa, J. Proenca, F. Arbab, Reconfiguration of Reo connectors triggered by dataflow, in: C. Ermel, J. de Lara, R. Heckel (Eds.), Proceeding of 7th International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT'08, in: Electronic Communications of the EASST, vol. 10, 2008.
[8] D. Clarke, A basic logic for reasoning about connector reconfiguration, Fundamenta Informaticae 82 (4) (2008) 361–390.
[9] D. Clarke, D. Costa, F. Arbab, Connector colouring I: Synchronisation and context dependency, Science of Computer Programming 66 (3) (2007) 205–225. doi:10.1016/j.scico.2007.01.009.
[10] M. Bonsangue, D. Clarke, A. Silva, Automata for context-dependent connectors, in: Coordination Models and Languages, in: Lecture Notes in Computer Science, vol. 5521, Springer Verlag, 2009, pp. 184–203. doi:10.1007/978-3-642-02053-7_10.
[11] G. Taentzer, AGG: A graph transformation environment for modeling and validation of software, in: J. Pfaltz, M. Nagl, B. Böhlen (Eds.), Application of Graph Transformations with Industrial Relevance, AGTIVE'03, in: Lecture Notes in Computer Science, vol. 3062, Springer, 2004, pp. 446–453. doi:10.1007/b98116.
[12] A. Rensink, The GROOVE simulator: A tool for state space generation, in: J. Pfalz, M. Nagl, B. Böhlen (Eds.), Applications of Graph Transformations with Industrial Relevance, AGTIVE'03, in: Lecture Notes in Computer Science, vol. 3062, Springer, 2004, pp. 479–485. doi:10.1007/b98116.
[13] A. Rensink, Explicit state model checking for graph grammars, in: R. Nicola, P. Degano, J. Meseguer (Eds.), Concurrency, Graphs and Models, in: Lecture Notes in Computer Science, vol. 5065, Springer Verlag, Berlin, 2008, pp. 114–132. doi:10.1007/978-3-540-68679-8_8.
[14] Eclipse coordination tools, http://reo.project.cwi.nl.
[15] ReoLive coordination web service, http://reo.project.cwi.nl/live.
[16] S. Klüppelholz, C. Baier, Symbolic model checking for channel-based component connectors, Electronic Notes in Theoretical Computer Science 175 (2) (2007) 19–37. doi:10.1016/j.entcs.2007.03.003.
[17] D. Plump, Hypergraph rewriting: Critical pairs and undecidability of confluence, in: M. Sleep, M. Plasmeijer, M. van Eekelen (Eds.), Term Graph Rewriting: Theory and Practice, 1993, pp. 201–213.
[18] R. Heckel, J.M. Küster, G. Taentzer, Confluence of typed attributed graph transformation systems, in: ICGT '02: Proceedings of the First International Conference on Graph Transformation, Springer-Verlag, London, UK, 2002, pp. 161–176.
[19] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems, TOPLAS 8 (2) (1986) 244–263. doi:10.1145/5397.5399.
[20] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, Compositional semantics for open Petri Nets based on deterministic processes, Mathematical Structures in Computer Science 15 (2005) 1–35.
[21] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, B. König, Bisimilarity and behaviour-preserving reconfigurations of open petri nets, CoRR abs/0809.4115, 2008. doi:10.2168/LMCS-4(4:3)2008.
[22] W.M.P. van der Aalst, The application of Petri Nets to workflow management, Journal of Circuits, Systems, and Computers 8 (1) (1998) 21–66. doi:10.1142/S0218126698000043.
[23] W.M.P. van der Aalst, Exterminating the dynamic change bug: A concrete approach to support workflow change, Information Systems Frontiers 3 (3) (2001) 297–317. doi:10.1023/A:1011409408711.
[24] R. Bruni, A.L. Lafuente, U. Montanari, E. Tuosto, Style-based architectural reconfigurations, Bulletin of the EATCS 94 (2008) 160–181.